UNIT-3
A8601

O O P

OBJECT ORIENTED PROGRAMMING

PRESENTED BY

M.YGANDHAR
Department of IT
Vardhaman College of Engineering

01
TOPIC
Exception-Handling Fundamentals

02
TOPIC
Exception Types, Using try catch

03
TOPIC
throw throws and finally keywords

04
TOPIC
Built-in Exceptions, Creating own exception subclasses

05
TOPIC
Multithreading: Life cycle o a thread

06
TOPIC
creating threads, thread priorities

07
TOPIC
Synchronizing threads

08
TOPIC
Interthread Communication.

Exception-Handling Fundamentals

- An exception is an **abnormal condition that arises** in a **code at run time**. In other words, an exception is a **runtime error**.
- When an **Exception occurs the normal flow** of the program is **disrupted** and the **program/Application terminates abnormally**, which is **not recommended**, therefore these **exceptions are to be handled**.
- An exception can occur for **many different reasons**, below given are some scenarios where exception occurs.
   i. Division by zero
   ii. Array out of bound access exception
   iii. A user has entered invalid data.
   iv. A file that needs to be opened cannot be found.
   v. A network connection has been lost in the middle of communications
- Some of **these exceptions** are **caused by user error**, others by **programmer error**, and others by **physical resources that have failed** in some manner.

Exception-Handling

- “Exception handling is the **mechanism to handle run time errors**, so that the **normal flow of application** can be **maintained.”**
- Exception Handling is **managed** by **using 5 Keywords.**

1. try
2. catch
3. throw
4. throws
5. finally

Types of Java Exceptions

There are mainly **three types of exceptio**

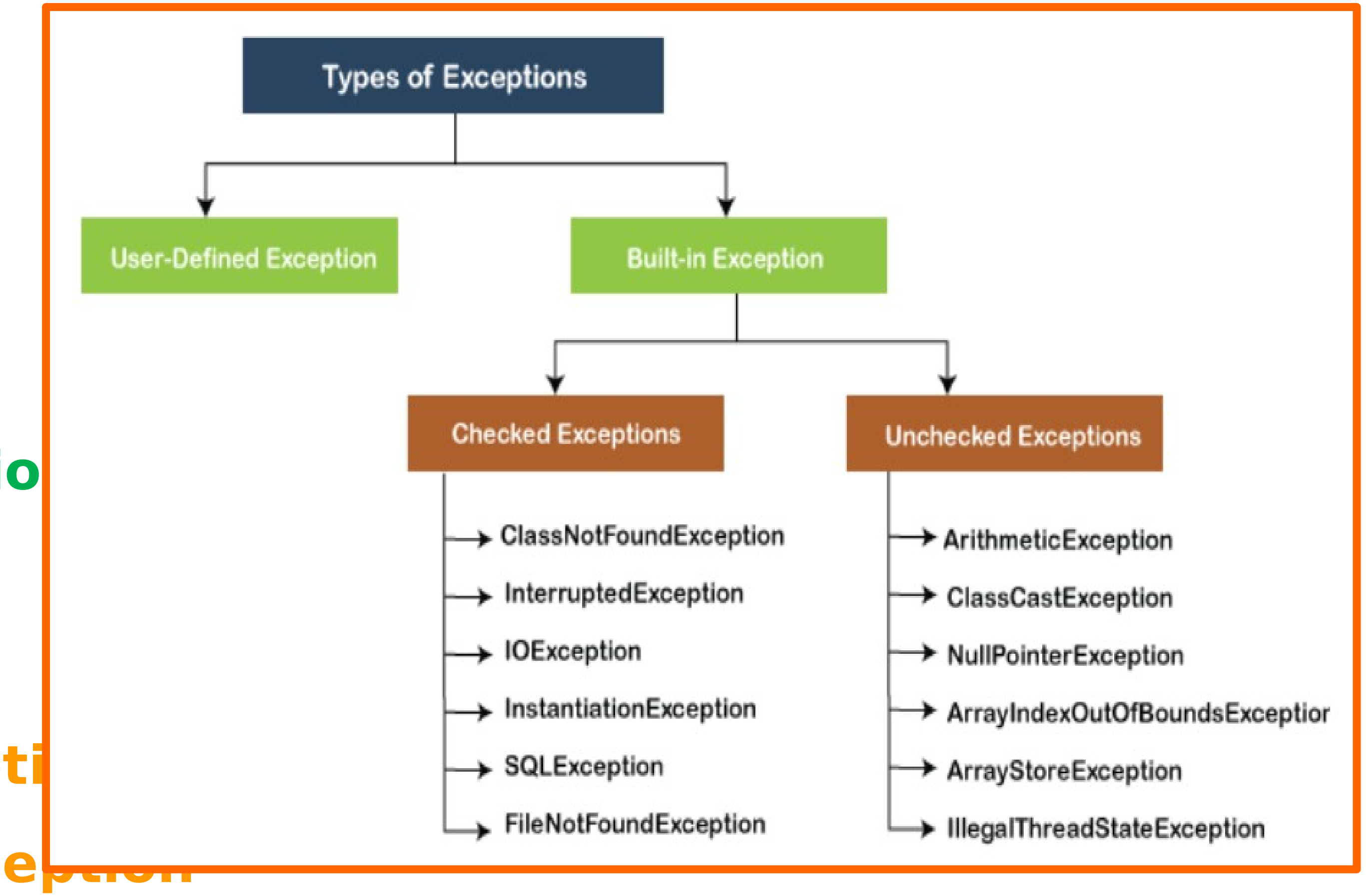
i. User Defined Exceptions

ii. Built in Exceptions

i. Checked Excepti

ii. Unchecked Exception

iii. Error



Types of Exceptions

i.Checked exceptions:

- A **checked exception** is an exception that **occurs at the compile time**, these are also called as **compile time exceptions.**
- These exceptions **cannot simply be ignored** at the time of compilation, the Programmer should take care of (handle) these exceptions.
- For example, if you **use FileReader class in your program to read data from a file**, if the **file specified** in its constructor **doesn't exist**, then an **FileNotFoundException** occurs, and compiler prompts the programmer to handle the exception.

i.Checked exceptions

1. ClassNotFoundException: This exception is thrown when the **JVM tries to load a class**, **which is not present** in the classpath.

```
Class.forName("oracle.jdbc.driver.OracleDriv
er");
```

2. FileNotFoundException: This exception is thrown when the **program tries to access a file** that **does not exist** or **does not open.** This error occurs mainly in file handling programs.

```
File file = new File("E:// file.txt");
FileReader fr = new
FileReader(file);
```

3.IOException: This exception is thrown when the input-output operation in a program fails or is interrupted during the program’s execution.

4.InterruptedException: This exception occurs **whenever a thread is processing, sleeping or waiting in a m** nd **it is interrupted.**

```
Thread t = new
Thread();
    t.sleep(10000);
```

Types of Exceptions

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo
{
     public static void main(String args[])
     {
          File file=new File("E://file.txt");
           FileReader fr = new
FileReader(file);
     }

}
```

If you try to compile the above program you will get exceptions as shown below.

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must
be  caught or declared to be thrown
 FileReader fr = new FileReader(file);
 ^
1 error
```

Some of the examples of checked exceptions

Types of Exceptions

ii.Unchecked exceptions:

- An **Unchecked exception** is an exception that **occurs at the time of execution**, these are also called as **Runtime Exceptions.**
- These include **programming bugs**, such as **logic errors** or improper use of an API.
- Runtime exceptions are **ignored at the time of compilation.**
- For example, if you have **declared an array of size 5 in your program**, and **trying to call the 6th element of the array** then an **ArrayIndexOutOfBoundsExceptionexception** occurs.
- To **guard against** and handle a **run-time error,** simply **enclose the code** that **you want to monitor inside a try block.**
- **Immediately** following the **try block, include a catch** clause that **specifies the exception type** that you wish to catch.

ii.Unchecked exceptions

1.ArithmeticException: This exception occurs when a **program encounters** an error in **arithmetic ope**

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(120/0);
    }
}
```

OUTPUT

```
D:\>java Test
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

2.ArrayIndexOutOfBoundsException: This exception is thrown when an **array is accessed using an illegal index.** The index used is either **more than the size of the array** or is

port n

```
class Test
{
    public static void main(String[] args)
    {
        int[] a = {10,20,30};
        System.out.println(a[50]);
    }
}
```

OUTPUT

```
D:\>java Test
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 50 out of bounds for length 3
```

ii.Unchecked exceptions

3.NullPointerException: This exception is raised when a **null object is referred** to in a program. NullPointerException is the most important and common exception in Java.

```
class Test
{
     public          static          void
main(String[] args)
     {
          String s = null;

System.out.println(s.length());
     }
}
```

OUTPUT

```
D:\>java Test
Exception          in          thread          "main"
java.lang.NullPointerException:
Cannot     invoke     "String.length()"     because
"<local1>" is null
        at Test.main(Test.java:6)
```

4.NumberFormatException: This exception is thrown when a method could not convert a string into a n

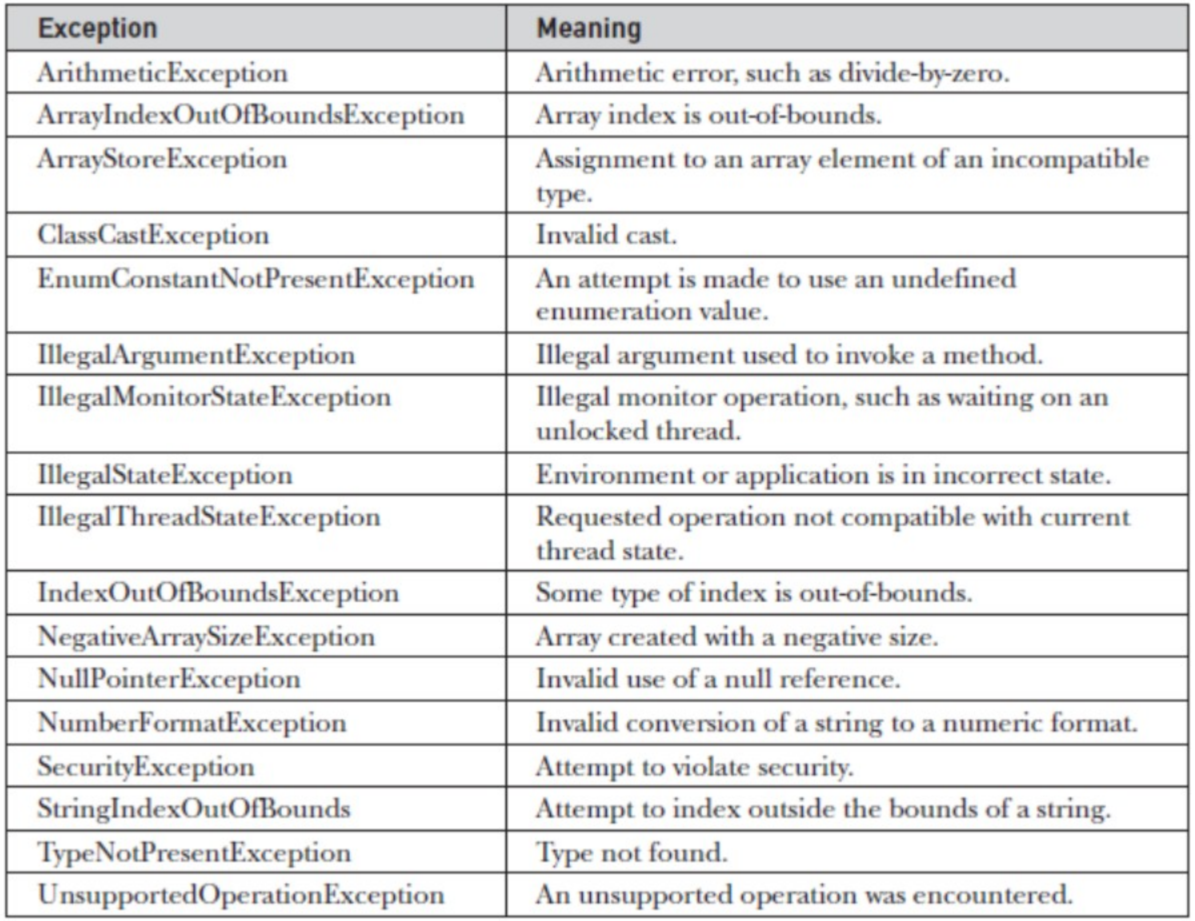
```
class Test
{
     public          static          void
main(String[] args)
     {
          String a = "abc";
          int
num=Integer.parseInt(a);
     }
}
```

ii.Unchecked exceptions

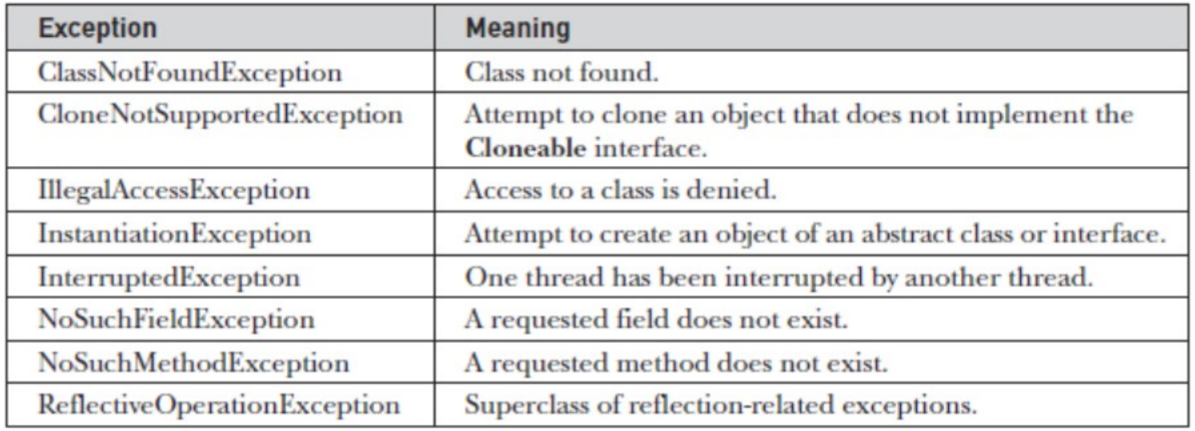
5. StringIndexOutOfBoundsException: This exception is thrown by the string class, and it indicates that the index is beyond the size of the string object or is negative.

```
class Test
{
     public static void main(String[] args)
     {
          String a = "JAVA";
          char c = a.charAt(6); // accessing 6
index element
          System.out.println(c);
     }
}
```

ii.Unchecked exceptions



Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.



Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Printing Exception information

The following are the methods to fetch exception information:

i. obj.stackTrace() ---------------> Name of the Exception: Description -> StackTrace

ii. obj.toString() -----------------> Name of the Exception: Description

iii. obj.getMessage() -------> Description

```
class Test
{
        public         static         void
main(String[] args)
        {
                int a[] = {11,22};
                try
                {

System.out.println(a[9]);
                }
                catch (Exception e)
                {

e.printStackTrace();
                }
        }
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 2
        at Test.main(Test.java:8)
```

```
class Test
{
        public          static          void
main(String[] args)
        {
                int a[] = {11,22};
                try
                {

System.out.println(a[9]);
                }
                catch (Exception e)
                {

System.out.println(e.toString());
                }
        }
}
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 2
```

```
class Test
{
        public static void main(String[]
args)
        {
                int a[] = {11,22};
                try
                {

System.out.println(a[9]);
                }
                catch (Exception e)
                {

System.out.println(e.getMessage())
;
                }
        }
}
```

```
Index 9 out of bounds for length 2
```

USING TRY AND CATCH

- This is the general form of an exception-handling block:

```
try
{
        ---------------// block of code to monitor for errors
 }
catch (ExceptionType1 exOb)
{
        ---------------// exception handler for ExceptionType1
}
```

USING TRY AND CATCH

- **ExceptionType** is the **type of exception** that has **occurred.**
- Program **Statements that we monitor** for exceptions are **contained in try block.**
- If an **exception occurs in try block**, it is **thrown.**
  - ✓ The **code can catch this exception using catch block** and **handle the exception** in a rational manner.
  - ✓ **System generated exceptions** are **automatically thrown** by the Java **runtime.**
  - ✓ **Catch block** is used to handle the exception, called **Exception Handler.**
  - ✓ Must be **used after try block only**. We can use **multiple catch blocks with a single try block.**
- The **throw keyword** is **used to manually throw an exception**
- **Any code** that absolutely **must be executed after a try block** completes is **put in finally block.**

```
class Exc2
 {
    public static void main(String args[])
    {
        int d, a;
        try
        {
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        }
        catch (ArithmeticException e)
        {
            System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
    }
}
```

Output

```
Division by zero.
After catch statement
```

MULTIPLE CATCH CLAUSES

- In some cases, **more than one exception** could be **raised** by a **single piece of code.**
- **To handle this type of situation**, **you can specify two** or **more catch clauses**, each catching a different type of exception.
- When an **exception is thrown**, each **catch statement is inspected in order**, and the first one **whose type matches that** of the **exception** is **executed.**
- After **one catch statement ex** d **execution continues after** the **try/catch**
- Syntax of Multiple catch stateme

```
try
{
        // block of code to monitor for errors
 }
catch (ExceptionType1 exOb)
{
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
        // exception handler for ExceptionType2
}
```

MULTIPLE CATCH CLAUSES

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[])
{
      try
      {
            int a = 10;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
      }
      catch(ArithmeticException e)
       {
            System.out.println("Divide by 0: " + e);
      }
      catch(ArrayIndexOutOfBoundsException e)
       {
            System.out.println("Array index oob: " + e);
      }
      System.out.println("After try and catch blocks.");
}
}
```

Output

```
a = 10
Array index oob:
java.lang.ArrayIndexOutOfBoundsEx
ception: Index 42 out of bounds for
length 1
After try and catch blocks.
```

NESTED TRY STATEMENTS

- The **try statement can be nested**.
- That is, **a try statement** can be **inside** the block of **another try.**
- If an **inner try statement does not** have a **catch handler for a particular exception**, the stack is unwound and the **next try statement's catch handlers are inspected for a match.**
- This **continues until one** of **the catch statements succeeds**, or **until the entire nested try statements are exhausted.**
- If **no catch statement matches**, then the **Java run-time system will handle the exception**. Here is an example that uses nested try statements:

```
// An example of nested try statements.
class NestTry
 {
public static void main(String args[])
{
    try
    {
        int a[] = {1,2,3,0,4};
        try
         {
            int b=a[2]/a[3];
        }
        catch(ArithmeticException  e)
        {
                System.out.println( e);
        }
        a[20]=44;

    }
    catch(ArrayIndexOutOfBoundsException e)
    {
         System.out.println(e);
    }
} }
```

```
OUTPUT:
C:\>java NestTry
java.lang.ArithmeticException: / by
zero
java.lang.ArrayIndexOutOfBoundsEx
ception
```

THROW

- So far, **you have only been catching exceptions** that are **thrown by the Java run-time system.**
- However, it is **possible for your program** to **throw an exception explicitly**, **using** the **throw statement**
- **throw keyword** is **used** to **explicitly throw** an **exception from a method** or constructor.
- **We can throw** either **checked** or **unchecked exceptions** in java by throw keyword.
- The **"throw" key-word** is mainly **us** **exception.**
- When a **throw statement is enc** **cution is halted**, and the **nearest catch statement** is **searched** for a **matching kind of exception.**
- The general form of throw is sh

Syntax

```
throw ThrowableInstance;
```

Example-1

```
throw new ArithmeticException( );
```

Example-2

```
throw new ArithmeticException(“Something went wrong!”);
```

Here, **ThrowableInstance** must be an **object of type Throwable** or a **subclass of Throwable.**

```
// Demonstrate throw.
class Test
{
     static void avg()
     {
           try
           {
                 throw new ArithmeticException("demo");
          }
          catch(ArithmeticException e)
          {
                System.out.println("Exception caught“+e);
          }
     }

      public static void main(String args[])
     {
             avg();
     }
}
```

Output

```
Exception caught
java.lang.ArithmeticException: demo
```

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
             Scanner s = new Scanner(System.in);
            System.out.println("Please enter your roll number");
             int roll = s.nextInt();
             try
         {
                 if (roll < 0)
            {
                        throw new ArithmeticException("The number entered is not positive");
                }
             else
            {
                        System.out.println("Valid roll number");
                }
             }
        catch (ArithmeticException e)
        {
                System.out.println(e.getMessage());
            }
     }
```

Output

```
Please enter your roll
number
1201
Valid roll number
```

Output

```
Please enter your roll
number
-27
The number entered is not
positive
```

THROWS

- In Java,**Methods may throw exceptions during the execution** of the program **using** the **throws keyword**.
- The **throws keyword** is **used** to declare the **list of exception that a method may throw** during execution of program.
- so that **anyone calling that method gets** a **prior knowledge** about which **exceptions** are to be handled.
- ys

Syntax

```
<return_type>   <method_name>   (   )   throws     <excpetion_name1>,
<exception_name2>
 {
     // body of method
}
```

Example

```
void Display( ) throws ArithmeticException, NullPointerException
{
     //code
}
```

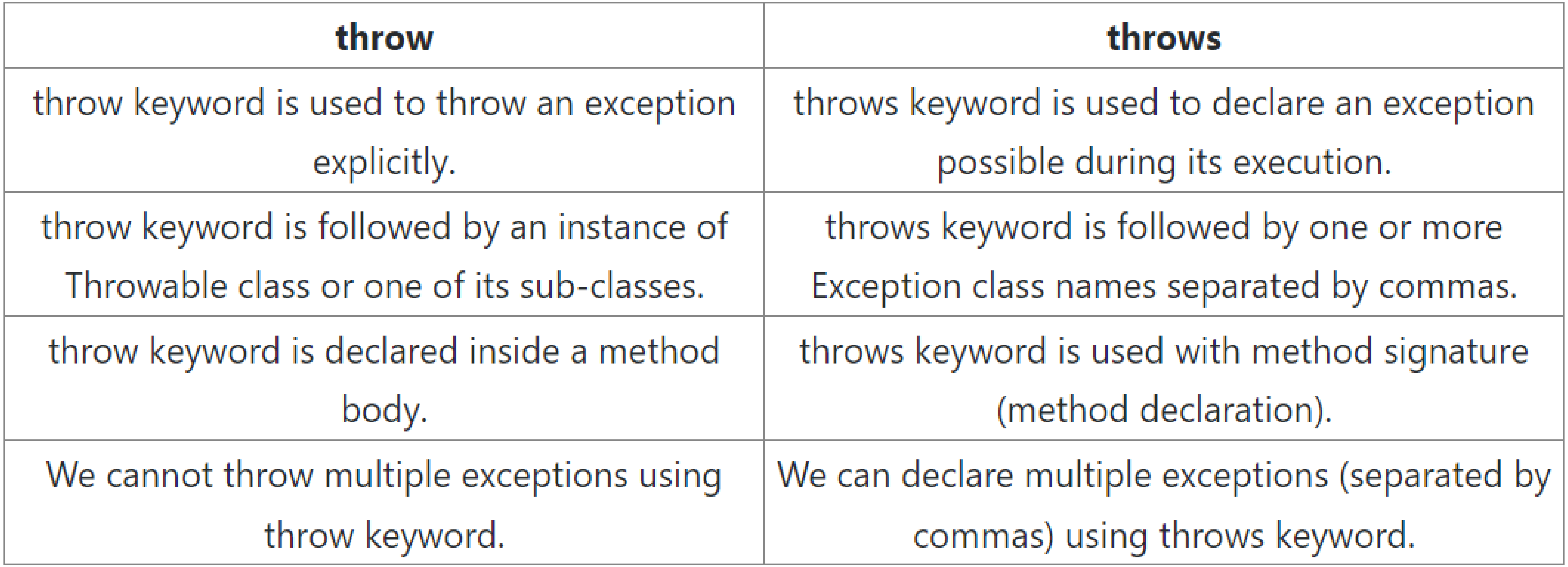
```
class Test
{
    static void check() throws ArithmeticException
    {
        System.out.println("Inside check function");
        throw new ArithmeticException("demo");
    }

    public static void main(String args[])
    {
         try
        {
            check();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught" + e);
         }
     }
}
```

OUTPUT

```
Inside check function
caughtjava.lang.ArithmeticException
: demo
```

DIFFERENCE BETWEEN THROW AND THROWS JAVA



throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

FINALLY

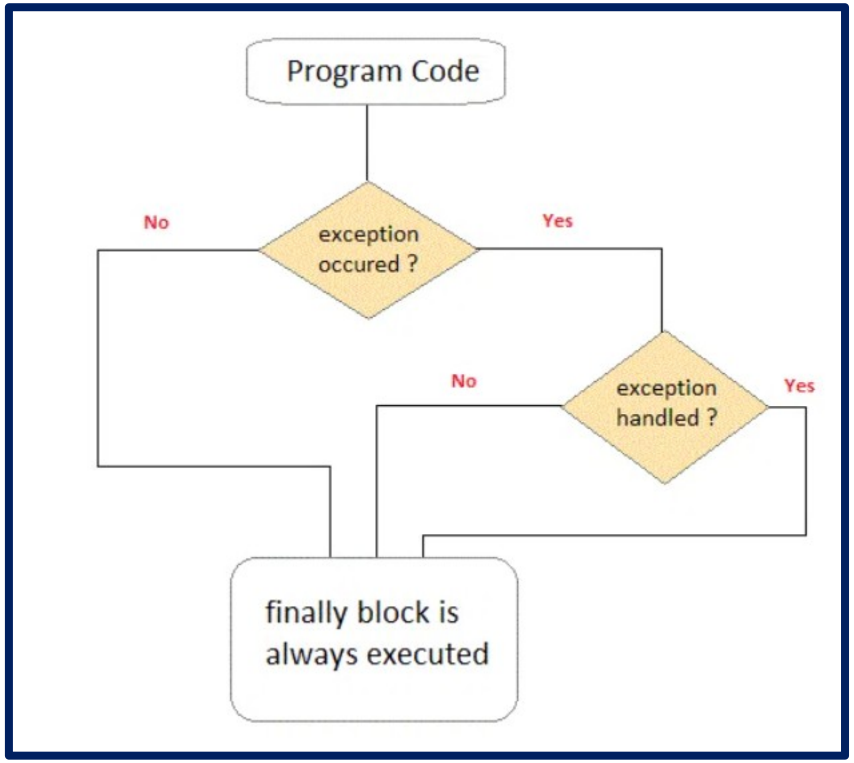
- A **finally** keyword is **used to create a block of code** that **follows a try block.**
- A finally block of code is **always executed whether an exception has occurred or not.**
- Using a finally block, it lets you run **any cleanup type statements** that you want
- **t th**

Syntax

```
try
{
      statement1;
     statement2;
}
finally
{
     statements;
}
```

Syntax

```
try
{
      statement1;
     statement2;
}
catch(Exceptiontype
e1)
{
     statement3;
}
finally
{
     statement4;
}
```



```
// Demonstrate finally.
class Demo
{
    public static void main(String[] args)
    {
        int a[] = new int[2];
         try
        {
                System.out.println("Access invalid element"+ a[3]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Exception caught");
         }
         finally
         {
                System.out.println("finally is always executed.");
        }
    }
}
```

OUTPUT

```
Exception caught
finally is always
executed.
```

```
// Demonstrate finally.
class FinallyDemo
 {
       static void procA()throws ArithmeticException
        {
              try
              {
                     System.out.println("inside procA");
                     throw new ArithmeticException("demo");
              }
              finally
              {
                      System.out.println("procA's finally");
              }
       }
       static void procB()
       {
              try
              {
                      System.out.println("inside procB");
                      return;
              }
              finally
              {
                        System.out.println("procB's finally");
              }
       }
```

```
public static void main(String args[])
{
       try
       {
                procA();
       }
       catch (Exception e)
       {
               System.out.println("Exception caught");
       }
       procB();

}
}
```

OUTPUT

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
```

Creating own exception

- Java **allows us to create** our **own exception class** to provide own exception implementation.
- These type of exceptions are **called user-defined exceptions** or **custom exceptions.**
- You can **create your own exception** simply **by extending** java **Exception class.**
- You can **define a constructor for your Exception** (not compulsory) and you can **override th** **customized message** on catch.

Syntax

```
class <name of the class> extends
Exception
{
    public String toString()
    {
        Statements;
    }
}
```

Creating own exception

Example

```
class MyException extends Exception
{
    String str1;
    MyException(String s)
    {
        str1=s;
    }
    public String toString()
    {
        return ("MyException Occurred:
"+str1) ;
    }
}
```

```
class Test
{
    public static void main(String args[])
    {
        try
        {
                System.out.println("Starting of
try block");
                throw new MyException("This is My
error Message");
        }
        catch(MyException exp)
        {
            System.out.println("Catch
Block") ;
            System.out.println(exp) ;
        }
    }
}
```

OUTPUT

```
Starting of try block
Catch Block
MyException Occurred: This is My error
Message
```

Example

```
class InsufficientFundsException extends
Exception
{
     InsufficientFundsException(String s)
     {
          super(s);
     }
}
class Test
{
     public static void main(String[] args)
     {
          java.util.Scanner obj = new
java.util.Scanner(System.in);
          double balance = 10000.00;
          double amt = obj.nextDouble();
          try
          {
               if(amt<=balance)
               {
                    System.out.println("pls take the
cash");
                    balance = balance - amt;
               }
```

```
else
throw new InsufficientFundsException("No Balance
in your account");
}
catch (InsufficientFundsException e)
{
     System.out.println(e.getMessage());
}
finally
{
     System.out.println("Updated
Balance:"+balance);
     System.out.println("Pls take your card");
}
}
}
```

OUTPUT

Multi Threading

- Multi Threading is a **specialized form** of **multi tasking.**
- Multitasking is a **process of executing multiple tasks simultaneously**. We **use** multitasking to **utilize** the **CPU**. Multitasking can be **achieved in two ways:**
  - ✓ **Process-based Multitasking (Multiprocessing)**
  - ✓ **Thread-based Multitasking (Multithreading)**
- **Executing several tasks simultaneously** where **each task** is a separate **independent** process such type of **multitasking** is called **process based multitasking.**
- **Executing several tasks simultaneously** where **each task** is a **separate independent** part of the **same program**, is called **Thread based multitasking.** And **each independent part** is called a **"Thread"**
- **Multithreading** refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU.
- Multithreading is a feature in Java that **concurrently executes two or more parts**

Multi Threading

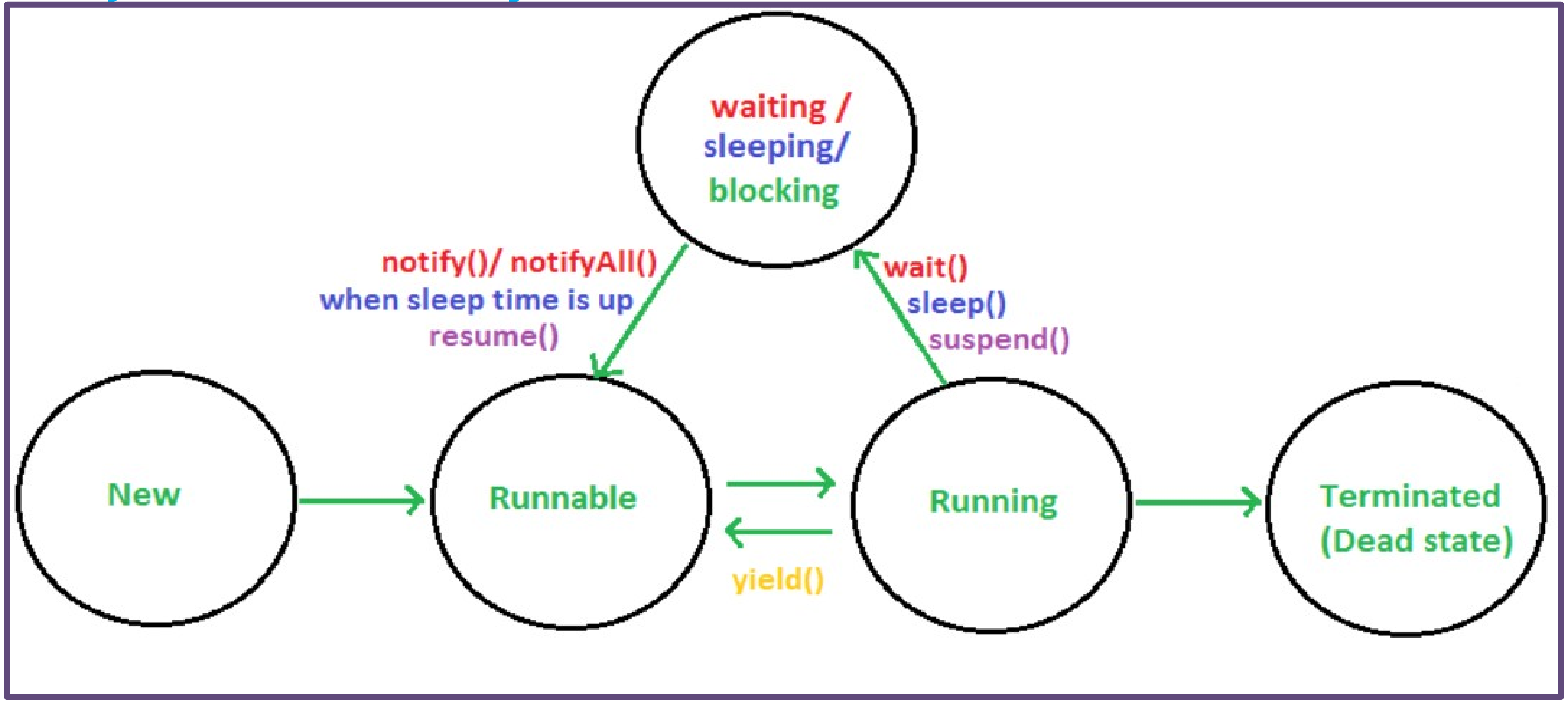
- A thread in Java is a ***lightweight process*** requiring **fewer resources.**
- **Each thread runs parallel** to each **other.**
- Java Provides **built in support** for **multi threaded programming.**
- Threads **share common memory** area.
- A **main program** is also single **thread.**

Advantages of MultiThreading

- **Allows** to write very **efficient program** that makes **maximum use of CPU.**
- Throughput – **Amount** of **work done** in **unit time increase.**
- It is used to **save time** as **multiple operations** are **performed** concurrently.
- The **threads are independent**, so it **does not block the user** to perform **multiple operations at the same time.**
- Since threads are independent, **other threads don’t get affected** even if an **exception occurs** in a single thread.

Thread Life Cycle

- The **life cycle of a thread** in Java is **controlled by JVM.**
- During the **execution of thread**, it is in any of the following **five states.**
- Hence **Java thread life cycle** is defined in **five states.**



Thread Life Cycle

i. New: The thread is in new state **when the instance** of **thread class** is **created** but **before the calling of start()** method.

ii. Runnable: The **thread is in runnable state after** the **invocation start()** method, but the **thread scheduler** has **not selected** it to be the running thread. ***Any number of threads exists in this state.***

iii. Running: The thread is **in running state** when the **thread scheduler selects a thread for execution.**

iv. Waiting/blocked/sleeping: this is the state **when the thread is still alive** but is **not eligible currently to run.**

v. (Blocked) Terminated: A **thread** is in **terminated** or dead state when its **run() method exits.**

- We can **define a Thread** in the **following 2 ways.**

i. By extending Thread class.

Multi Threading

- The **main important application** areas of **multithreading are:**
  - To implement **multimedia graphics.**
  - To develop **animations.**
  - To develop **video games** etc.
  - To develop **web** and **application servers**.

Thread class:

- Java provides **Thread class** to **achieve thread programming.**
- **Thread class** provides **constructors** and **methods** to create and **perform operations on a thread.**
- **Commonly used Constructors** of Thread class:

  i. Thread() **ii. Thread(String name)**

  iii. Thread(Runnable r) **iv. Thread(Runnable r, String name)**

Commonly used methods of Thread class

- Commonly used methods of Thread class:

1. public void start():

- **starts the execution of the thread**. start() method of Thread class is used to start a newly created thread.

It performs following tasks:

1. A new thread starts

2. The **thread moves from New state** to the **Runnable state.**

3. When the **thread gets a chance to execute**, its target **run() method will be called** and executes.

2. public void run(): It is the **entry point of a thread** , used to **perform action** for a thread.

3. public int getPriority(): It returns the **priority of the thread**.

4. public int setPriority(int priority): It is used to **set** or **change the priority** of the

Commonly used methods of Thread class

6. public void setName(String name): It is used to **set** or **change** the **name of the thread.**

7.public static Thread currentThread(): returns the **reference** of **currently executing thread.**

8. Public static void sleep(long miliseconds): Causes the **currently executing thread to suspend** (temporarily cease execution) for the **specified number of milliseconds**.

9. public void join(): waits for a **thread to die/terminate.**

10.public boolean isAlive(): to **check** the **thread is still running**. Returns true if the thread is still running.

Creating Thread – Extending Thread Class

Step-1:

-Create a **new class** that **extends Thread**, and then **create an instance of that class.**

Step-2:

-The **extending class must override the run () method**, which is the **entry point of the new thread.**

-The **actual code** for the **thread** to execute **will be provided here.**

- **Once** the **run () method completes**, the **thread** will **die and terminate.**

Step-3:

-**Calls start () method** to begin **execution of new thread.**

Step-4:

-Call the **Thread class constructor** using **super keyword if necessary.**

Creating Thread – Extending Thread Class

```
class MyThread extends Thread
{
    public void run()
    {
         for(int i=0;i<5;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
public class TheadDemo1
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread();
        t1.start();
        for(int i=0;i<5;i++)
        {
            System.out.println("main thread");
        }
    }
}
```

OUTPUT

```
main thread
main thread
main thread
main thread
main thread
Child Thread
Child Thread
Child Thread
Child Thread
Child      Thread
```

Creating Thread – Implementing Runnable Interfa

Step-1:

-To **create a new Thread**, the **class must implements Runnable interface.**

Step-2:

-**Provide implementation** for the **only one method run (),** which is the entry point of newly created thread. -The **actual code** for **the thread** will be **mentioned here**.

- Once the **run () method completes**, the **thread will die** and **terminate.**

Step-3:

-**Instantiate an object of type Thread** within the **newly created thread class**. Thread(Runnable obj)

Step-4:

-Call the **start () method explicitly to start the thread**.

-It **makes a call to run() method** to start the execution.

Creating Thread – Implementing Runnable Inter

```
class MyRunnable implements Runnable
{
        public void run()
        {
               for(int i=0;i<5;i++)
               {    System.out.println("Child Thread");
               }
        }
}
class ThreadDemo
{
        public static void main(String[] args)
        {
              MyRunnable r=new MyRunnable();
                    Thread  t=new  Thread(r);//here  r  is  a  Target
Runnable
              t.start();
                es for(int i=0;i<10;i++)
              {System.out.println("main thread");
              }
         }
}
```

OUTPUT

```
main thread
main thread
main thread
main thread
main thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
```

Creating Mutliple Threads using Thread class

```
//Multithreading By Extending Thread class
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<3;i++)
		{
			System.out.println(i);
			try
			{
				Thread.sleep(1000);
			}
			catch(Exception e)
			{
				System.out.println(e);
			}
		}
	}
}
```

```
public class Own
{
	public static void main(String[] args)
	{

		Thread t = Thread.currentThread();
		System.out.println("ID:"+t);
		System.out.println("Name:"+t.getName());
		t.setName("Vardhaman");
		System.out.println("After renaming:"+t.getName());
		MyThread t1 = new MyThread();
		t1.start();
		try
		{
			t1.join();
		}
		catch(Exception e)
		{	System.out.println(e)
		MyThread t2 = new My
		t2.start();

	}
}
```

OUTPUT

```
ID:Thread[#1,main,5,
main]
Name:main
After
renaming:Vardhaman
0
1
2
0
1
2
```

Creating Mutliple Threads using Thread class

```
//Multithreading By Extending Thread class
class NewThread extends Thread
{
        NewThread(String threadname)
        {
            super(threadname);
            System.out.println("Child Thred -> " +this );
        }
        public void run()
        {
        try
         {
            for(int i = 5; i > 0; i--){
            System.out.println(getName ()+ ": " + i);
            sleep(500);}
        }
        catch (InterruptedException e)
        {
            System.out.println(getName()                  +
"Interrupted");
        }
         System.out.println(getName() + " Completed");
        } }
```

Creating Multiple Threads using Thread class

```
class MultiThread1
{
        public static void main(String args[])
        {
            Thread t = Thread.currentThread();
            System.out.println("Name is --->" +t.getName());
            System.out.println("Main ---> " +t);
            NewThread t1 = new NewThread("One");
            NewThread t2= new NewThread("Two");
            NewThread t3 = new NewThread("Three");
            t1.start();
            t2.start();
            t3.start();
            try
            {
                for(int n=1;n<=5;n++){
                System.out.println("Main Thread->" +n);
                Thread.sleep(500);
            }
            catch (InterruptedException e)
            {
                System.out.println("Main thread Interrupted");
            }
                System.out.println("Main thread Completed");
        } }
```

OUTPUT

```
Name is --->main
Main ---> Thread[#1,main,5,main]
Child            Thred            ->
Thread[#21,One,5,main]
Child            Thred            ->
Thread[#22,Two,5,main]
Child            Thred            ->
Thread[#23,Three,5,main]
Main Thread->1
One: 5
Three: 5
Two: 5
Main Thread->2
Two: 4
Three: 4
One: 4
One: 3
Main Thread->3
Two: 3
Three: 3
One: 2
Main Thread->4
Two: 2
Three: 2
Three: 1
Two: 1
Main Thread->5
One: 1
One Completed
Main thread Completed
Two Completed
```

Creating Mutliple Threads using Runnable Interface

```
class NewThread implements Runnable
{
        Thread t;
        NewThread()
        {
            t = new Thread(this, "Child Thread");
           System.out.println("Child thread: " + t);
                            System.out.println("The  thread  name  is
"+t.getName());
           t.start();
       }
public void run()
 {
          try
          {
                for(int i = 1; i <=10; i++)
                {
                             System.out.println("Child Thread: ->" +
t.getName() + ":" + i);
                    Thread.sleep(500);
                }
         }
       catch (InterruptedException e)
{
                System.out.println("Child interrupted.");
}}}
```

Creating Mutliple Threads using Runnable Interface

```
public class MultiThread2
{
     public static void main(String args[ ] )
     {
     NewThread t1 = new NewThread();
     NewThread t2 = new NewThread();
     NewThread t3 = new NewThread();
     try
     {
          for(int i = 5; i > 0; i--) {
          System.out.println("Main Thread: " + i);
          Thread.sleep(1000);
          }
          }
     catch (InterruptedException e){
     System.out.println("Main                  thread
interrupted.");
     }
     }
}
```

Thread Priorities

- **Thread priorities** are **used by** the **thread scheduler** to **decide which thread** should **be allowed to run.**
- A **higher-priority thread** get **more CPU time** than **lowerpriority thread.**
- **Thread class defines** the method **setPriority()** to **assign priority** to a **thread.**
- **final void setPriority(int level)** The **value of level** must be **within the range MIN_PRIORITY** and **MAX_PRIORITY. Currently**, these **values are 1 and 10**, respectively.
- To **return a thread** to **default priority**, specify **NORM_PRIORITY**, which is **currently 5.**
- These priorities are defined as static final variables within Thread.

  public static final int MIN_PRIORITY (1)

  public static final int MAX_PRIORITY (10)

  public static final int NORM_PRIORITY (5)

Thread Priorities

```
class NewThread extends Thread
 {          NewThread(String name)
            {     super (name);
            }
            public void run()
            {       System.out.println("Hello   -->"+   getName()   +   "-
>"+getPriority() ); }
}
 public class PriorityDemo
 {
             public static void main(String args[])
             {
                   Thread t = Thread.currentThread();
                   System.out.println("The   main   thread   priority   is:   "
+t.getPriority());
                   NewThread t1 = new NewThread("One");
                   NewThread t2 = new NewThread("Two");
                   NewThread t3 = new NewThread("Three");
                   t1.setPriority(Thread.MIN_PRIORITY);
                   t2.setPriority(Thread.MAX_PRIORITY);
                   t3.setPriority(5);
                   t1.start();
                   t2.start();
                   t3.start();
            }
```

OUTPUT

```
The main thread priority
is : 5
Hello -->Two->10
Hello -->Three->5
Hello -->One->1
```

Synchronizing Threads

- When **two or more threads need access** to a **shared resource**, they **need some way to ensure** that the **resource will be used by only one thread at a time**. The **process** by which this is **achieved** is **called synchronization.**
- Java **provides unique, language-level** support for it.
- Synchronization **allows only one thread** to **access a shared resource.**
- The **advantages of Synchronization** are:
  - ❖ To prevent thread interferences.
  - ❖ To prevent inconsistency of data.
- Thread **Synchronization can be of two forms**
  - **i. Mutual Exclusion**
  - **ii. Inter Thread communication.**

Synchronizing Threads

1. Mutual Exclusion

Keeps threads from interfering with one another while sharing data. **Allows only one thread to access shared data.** Mutual exclusive threads can be **implemented using**

i) Synchronized Method **ii) Synchronized Block**

i) Synchronized Method

- ✓ A **method defines as synchronized**, then the method is a synchronized method.
- ✓ Synchronized method is **used to lock an object for any shared resource.**
- ✓ When a **thread calls a synchronized method**, it **automatically acquires lock** for that object and **releases it whe**

The form of synchronized method is:

```
class <class Name>
{
                    synchronized                type
methodName(arguments)
     {
     //method body for Synchronization
     }
 }
```

Synchronizing Threads

```
//synchronized Method - for Synchronization
class Display
{
        public synchronized void wish(String name)
        {
                for(int i=0;i<3;i++)
                {
                        System.out.print("good morning:");
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e)
                        {}
                        System.out.println(name);
                }
        }
}
```

```
class MyThread extends Thread
{
        Display d;
        String name;
        MyThread(Display d2,String n2)
        {
                d=d2;
                name=n2;
        }
        public void run()
        {
                d.wish(name);
        }
}
```

```
class SynchronizedDemo2
{
        public static void main(String[] args)
        {
                Display d1=new Display();
                MyThread                                t1=new
MyThread(d1,"dhoni");
                MyThread                                t2=new
MyThread(d1,"yuvaraj");
                t1.start();
                t2.start();
        }
}
```

OUTPUT

```
good
morning:dhoni
good
morning:dhoni
good
morning:dhoni
good
morning:yuvaraj
good
morning:yuvaraj
good
morning:yuvaraj
```

Synchronizing Threads

```
//synchronized Method - for Synchronization
class Table
{
        synchronized void printTable(int n)
       {
                try{
                      for(int i=1;i<=10;i++)
                      {

       System.out.println(n+"x"+i+"="+n*i);
                      Thread.sleep(1000);
                      }
               }
       catch(InterruptedException e)
       {     System.out.println(e);
       }
       }
 }
class MyThread1 extends Thread
{
        Table t;
       MyThread1(Table tab)
        {
       t=tab;
        }
       public void run()
       {
        t.printTable(8);
        }
}
```

```
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table tab)
        {
         t=tab;
         }
        public void run()
        {
        t.printTable(9);
         }
 }
public class SyncDemo1
 {
          public   static   void   main(String
args[])
        {
        Table obj = new Table();
                    MyThread1          t1=new
MyThread1(obj);
                    MyThread2          t2=new
MyThread2(obj);
        t1.start();
        t2.start();
         }
 }
```

OUTPUT

```
8x1=8
8x2=16
8x3=24
8x4=32
8x5=40
8x6=48
8x7=56
8x8=64
8x9=72
8x10=80
9x1=9
9x2=18
9x3=27
9x4=36
9x5=45
9x6=54
9x7=63
9x8=72
9x9=81
9x10=90
```

Synchronizing Threads

ii) Synchronized Block:

- ✓ Synchronized block can be used to **perform synchronization** on **any specific resource of a method.**
- ✓ Synchronized block is **used to lock** an object for **any shared resource.**
- ✓ **Scope of Synchronized block** is **smaller than** the **method.**

The form of synchronized block is

```
synchronized(object)
{
    //Statements to be synchronized
}
```

Synchronizing Threads

```
//Thread Synchronization using - synchronized block
class Table
{
           void printTable(int n)
          {
          System.out.println(“I am non synched stmt");
          synchronized(this)
           {
                   for(int i=1;i<=10;i++)
                   {
                            System.out.println(n*i);
                            try{    Thread.sleep(1000);
                                  }
                            catch(InterruptedException e)
                            {    System.out.println(e);
                            }
                   }
          } //end of sync block
          System.out.println(" I am last non synched stmt ");
          } //end of the method
 }
class MyThread1 extends Thread
{         Table t;
          MyThread1(Table tab)
          {
                   t=tab;
          }
          public void run()
          {
                   t.printTable(5);
          }
 }
```

```
class MyThread2 extends Thread
{
          Table t;
          MyThread2(Table tab)
          {
                   t=tab;
          }
          public void run()
          {
                   t.printTable(100);
          }
}
public class SyncDemo2
 {
          public static void main(String args[])
           {
                   Table obj = new Table();
                   MyThread1 t1=new MyThread1(obj);
                   MyThread2 t2=new MyThread2(obj);
                   t1.start();
                   t2.start();
          }
}
```

InterThread Communication

- **Inter-thread communication** in Java is a **technique through which multiple threads communicate** with **each other.**
- It provides **an efficient way** through **which more than one thread communicate** with **each other** by **reducing CPU idle time**.
- When **more than one threads are executing simultaneously**, **sometimes they need to communicate** with **each other** by exchanging information with each other.
- A **thread exchanges information before** or **after** it **changes its state.**
- There are **several situations** where **communication** between threads is **important.**
- For example, suppose that there are **two threads A and B.**
- **Thread B uses data produced by Thread A** and performs its task.

InterThread Communication

- If **Thread B waits for Thread A to produce data**, it will **waste many CPU cycles**. But **if threads A and B communicate with each other when they have completed their tasks**, they **do not have to wait** and **check each other's status** every time.
- This **type of information exchanging between threads** is called **inter-thread communication in Java.**
- Inter thread communication in Java can be **achieved by using three methods** provided by Object class of java.lang package. They are:
  1. **wait()**
  2. **notify()**
  3. **notifyAll()**
- **These methods** can be **called only** from **within a synchronized method** or **synchronized block** of code **otherwis**e, an exception named

InterThread Communication

i.wait():

-This method is **used to make the particular Thread wait until it gets a notification**.

-This method **pauses the current thread** to the **waiting room dynamically.**

ii. notify():

-This method is **used to send the notification to one of the waiting thread** so that **thread enters** into a **running state** and **execute the remaining task**.

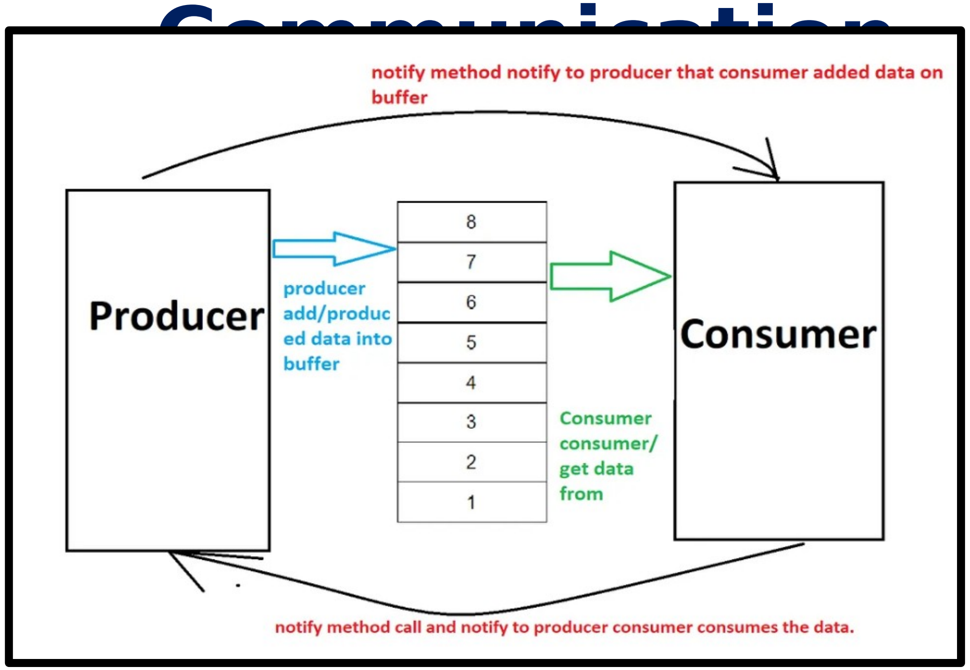
-This method **wakeup a single thread** into the **active state.**

iii. notifyAll():

-This method is used to **send the notification to all the waiting threads** so that **all thread enters into the running state** and execute simultaneously.

-This method **wakes up all the waiting threads** that act on the common objects.

Producer Consumer Problem for Interthread Communication


notify method notify to producer that consumer added data on buffer
Producer
producer add/produced data into buffer
8
7
6
5
4
3
2
1
Consumer consumer/ get data from
Consumer
notify method call and notify to producer consumer consumes the data.

- Producer Thread- Produce items to Buffer (Add Items) Consumer Thread- Consume items from Buffer (Removes Items).
- The **two conditions for Producer – Consumer** Problem is :

  1. Producer cannot add an item into a buffer if it is full

  2. Consumer cannot consume an item if it is empty.
- If **no communication**, these two conditions are not satisfied then the **CPU is always in polling (loop).**

Producer Consumer Problem for Interthread Communication

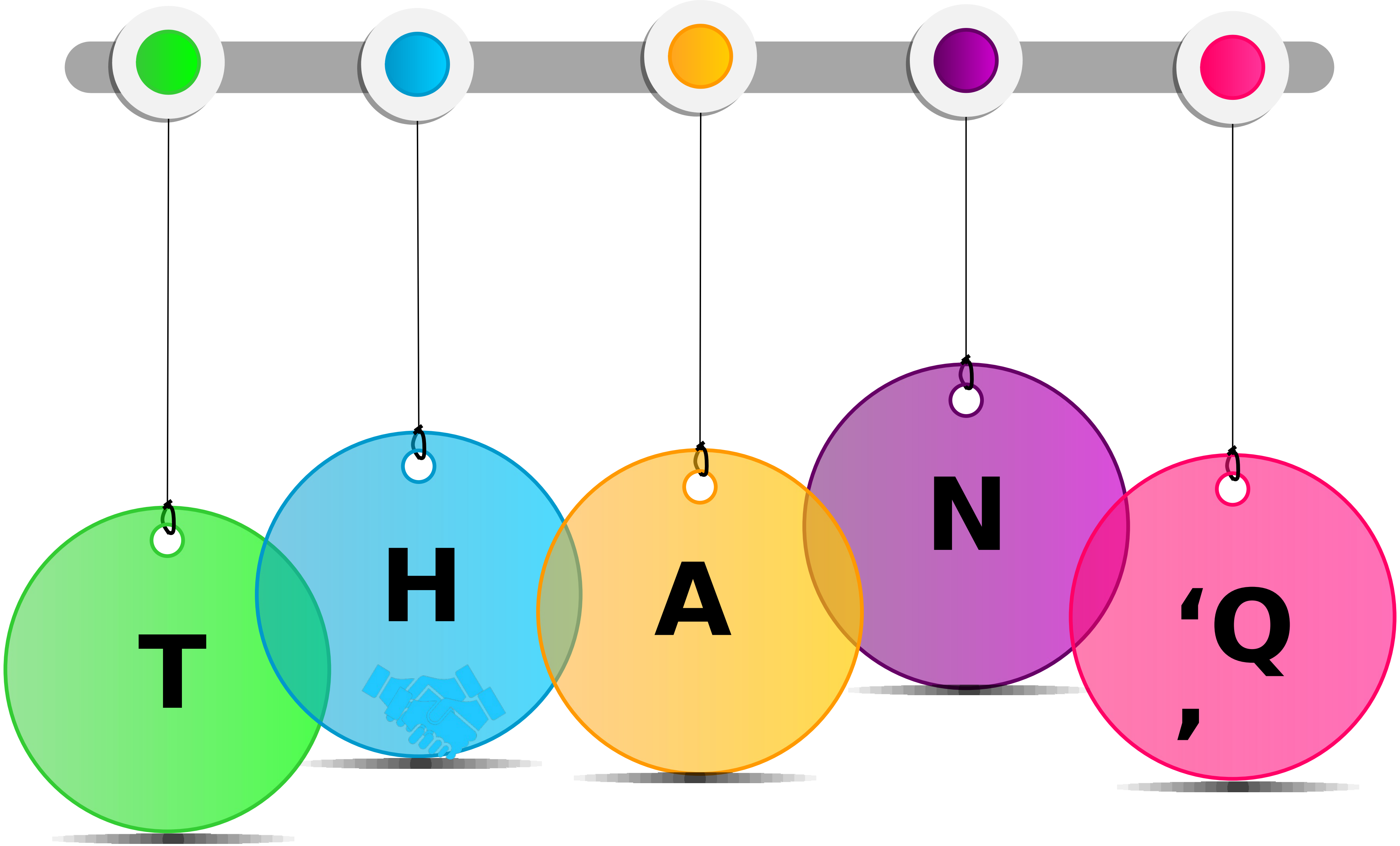
```
//Producer-Consumer    problem    --->   Inter    Thread
Communication.
class Buffer
{
        int item;
        boolean produced = false;
        synchronized void produce(int x)
        {
                if(produced)
                {
                        try{
                        wait();
                        }
                        catch(InterruptedException ie)
                        {
                                System.out.println("Exception Caught");
                        }
                }
                item =x;
                System.out.println("Producer    -    Produced-->"
+item);
                produced =true;
                notify();
        }
```

```
synchronized int consume()
{
        if(!produced)
        {
                try{
                wait();
                }
                 catch(InterruptedException ie)
                {
                                System.out.println("Exception
Caught " +ie);
                }
        }
        System.out.println("Consumer - Consumed "
+item);
        produced = false;
        notify();
        return item;
         }
}
```

Producer Consumer Problem for Interthread Communication

```
class Producer extends Thread
{
        Buffer b;
        Producer( Buffer b)
        {
                this.b = b;
                start();
        }
        public void run()
        {
                 b.produce(10);
                b.produce(20);
                b.produce(30);
                b.produce(40);
                 b.produce(50);
        }
}
```

```
class Consumer extends Thread
{
        Buffer b;
         Consumer(Buffer b)
         {
                this.b = b;
                start();
        }
        public void run()
         {
                 b.consume();
                 b.consume();
                b.consume();
                 b.consume();
        }
}
public class PCDemo
{
                public static void main(String args[])
                {
                        Buffer b = new Buffer(); //Synchronized Object
                        Producer p = new Producer(b);
                        Consumer c = new Consumer(b);
                }
}
```



T H A N ‘Q ,